WHAT IS CLAIMED IS:

1. A method of detecting mutability of variables, objects, fields, and classes in a program component, said component being written in an object-oriented programming language, comprising the steps of:

determining whether any variable in the program component could undergo a state modification of a first type, said first type state modification being made by at least one method that is within the program component; and

performing encapsulation analysis to determine whether any variable in the program component could undergo a state modification of a second type, said second type state modification being made by at least one method that is not within the program component;

wherein a variable is mutable if its state ever changes after said variable is initialized, the state of said variable being its value together with a state of any referenced object;

wherein an object is mutable if its state ever changes after said object is initialized, said state of said object being a set of states of all associated variables;

wherein a field is mutable if any variable corresponding to said field is mutable; and

wherein a class is mutable if any instance fields implemented by said class are mutable.

2. The method as recited in claim 1, the first type state modification determination step comprising the steps of:

detecting possible first type state modification of a value held in said each variable; and

detecting possible first type state modification of a state of any object referenced by said each variable.

3. The method as recited in claim 1, the encapsulation analysis step comprising the steps of:

detecting possible second type state modification of a value held in said each variable;

detecting possible second type state modification of a state of any object referenced by said each variable, said possible second type state modification of a state of any object occurring at a point of initialization; and

detecting possible breakage of variable encapsulation;

wherein a variable is encapsulated if all references to objects reachable from it are defined within the program component; and

wherein variable encapsulation is broken if a method within the program component causes a mutable object reachable from the variable to become accessible to at least one method that is not within the program component.

4. The method as recited in claim 1, wherein the method is implemented in a Java environment, said any instance fields being non-`static` fields, said variables being class variables or instance variables, each of said class variables being initialized upon completion of its corresponding `<clinit>` method, and each of said instance variables being initialized upon completion of its corresponding `<init>` method.

5. The method as recited in claim 1, further comprising the step of:

identifying isolation faults due to detected mutable global variables or objects.

6. The method as recited in claim 1, further comprising the step of:

identifying fields and objects that can be determined to be constants because said identified fields and objects are not in the set of detected mutable fields and objects.

7. A method of detecting mutability of classes in a program component, said component being written in an object-oriented programming language, comprising the steps of:

obtaining a set of classes, each of said classes being classified as one of mutable, immutable, and undecided;

testing each undecided class, said test being comprised of the sub-steps of:

testing each field in said undecided class being tested, said test being comprised of the sub-sub-steps of:

determining whether any variable corresponding to said each field could undergo a state modification of first type, said first type state modification being made by at least one method that is within said component; and

performing encapsulation analysis to determine whether any variable corresponding to said each field could undergo a state modification of a second type, said second type state modification being made by at least one method that is not within said component;

classifying said each field as immutable if no possible first type or second type state modifications are found;

classifying said each field as undecided if there is insufficient class mutability information; and

classifying said each field as mutable otherwise;

re-classifying said undecided class as mutable if any fields in said undecided class are mutable;

re-classifying said undecided class as immutable if all fields in said undecided class are immutable;

Docket No. YOR920000253US1

repeating said testing each undecided class step until a number of undecided classes after a repetition of said testing step is identical to a number of undecided classes before the repetition of said testing step; and

re-classifying remaining undecided classes as mutable classes.

8. A method of detecting mutability of classes in a program component, said component being written in an object-oriented programming language, comprising the steps of:

obtaining a set of classes, each of said classes being classified as one of mutable, immutable, and undecided;

testing each undecided class, said test being comprised of the sub-steps of:

testing each instance field in said undecided class being tested, said test being comprised of the sub-sub-steps of:

determining whether any variable corresponding to said each instance field could undergo a state modification of first type, said first type state modification being made by at least one method that is within said component; and

performing encapsulation analysis to determine whether any variable corresponding to said each instance field could undergo a state modification of a second type, said second

type state modification being made by at least one method that is not within said component;

classifying said each instance field as immutable if no possible first type or second type state modifications are found;

classifying said each instance field as undecided if there is insufficient class mutability information; and

classifying said each instance field as mutable otherwise;

re-classifying said undecided class as mutable if any instance fields in said undecided class are mutable;

re-classifying said undecided class as immutable if all instance fields in said undecided class are immutable;

repeating said testing each undecided class step until a number of undecided classes after a repetition of said testing step is identical to a number of undecided classes before the repetition of said testing step; and

re-classifying remaining undecided classes as mutable classes.

9. The method as recited in claim 8, the first type state modification determination sub-sub-step comprising the steps of:

detecting possible first type state modification of a value held in said each variable; and

detecting possible first type state modification of a state of any object referenced by said each variable;

wherein a state of an object is modified if it can change after said object is initialized, and the state of an object is a set of states of all associated variables; and

wherein a variable is mutable if its state ever changes after said variable is initialized, the state of said variable being its value together with a state of any referenced object.

10. The method as recited in claim 8, the performing encapsulation analysis sub-sub-step comprising the steps of:

detecting possible second type modification of a value of said each variable;

detecting possible second type modification of a state of any object referenced by said each variable, said possible second type state modification of a state of any object occurring at a point of initialization; and

detecting possible breakage of variable encapsulation;

wherein a state of an object is modified if it can change after said object is initialized, and the state of an object is a set of states of all associated variables;

wherein a variable is mutable if its state ever changes after said variable is initialized, the state of said variable being its value together with a state of any referenced object;

wherein a variable is encapsulated if all references to objects reachable from it are defined within said component; and

wherein variable encapsulation is broken if a method within the program component causes a mutable object reachable from a variable to become accessible to at least one method that is not within said component.

11. The method as recited in claim 8, wherein the method is implemented in a Java environment, said each variable corresponding to said each instance field being a non-`static` variable, and each non-`static` variable being initialized upon completion of its corresponding `<init>` method.

12. The method as recited in claim 8, further comprising the steps of:

identifying an object as mutable if it is an instance of a mutable class;

identifying an object as immutable if it is an instance of an immutable class; and

identifying fields and objects that can be determined to be constants because said identified fields and objects are not in a set of detected mutable fields and objects.

13. The method as recited in claim 8, further comprising the step of:

testing mutability of each undecided class field in each class.

14. The method as recited in claim 13, further comprising the step of:

identifying isolation faults due to detected mutable class fields.

15. The method as recited in claim 13, the step of testing mutability of each undecided class field in each class comprising the sub-steps of:

determining whether any variable corresponding to said each undecided class field could undergo a first type state modification; and

performing encapsulation analysis to determine whether any variable corresponding to said each undecided class field could undergo a second type state modification.

16. The method as recited in claim 15, wherein the determining whether any variable corresponding to said each undecided class field could undergo a first type state modification sub-step comprises the steps of:

detecting possible first type state modification of a value held in said each variable; and

detecting possible first type state modification of a state of any object referenced by said each variable;

wherein a state of an object is modified if it can change after said object is initialized, and the state of an object is a set of states of all associated variables; and

wherein a variable is mutable if its state ever changes after said variable is initialized, the state of said variable being its value together with a state of any referenced object.

17. The method as recited in claim 15, wherein the performing encapsulation analysis to determine whether any variable corresponding to said each undecided class field could undergo a second type state modification sub-step comprises the steps of:

detecting possible second type state modification of a value of said each variable;

detecting possible second type state modification of a state of any object referenced by said each variable, said possible second type state modification of a state of any object occurring at a point of initialization; and

detecting possible breakage of variable encapsulation;

wherein a state of an object is modified if it can change after said object is initialized, and the state of an object is a set of states of all associated variables;

wherein a variable is mutable if its state ever changes after said variable is initialized, the state of said variable being its value together with a state of any referenced object;

wherein a variable is encapsulated if all references to objects reachable from it are defined within said component; and

wherein variable encapsulation is broken if a method within the program component causes a mutable object reachable from a variable to become accessible to at least one method that is not within said component.

18. The method as recited in claim 13, wherein the method is implemented in a Java environment, said variables corresponding to said each undecided class field being `static` variables, and each `static` variables being initialized upon completion of its corresponding `<clinit>` method.

19. A method of detecting mutability of classes and class variables in a program component, said component being written in an object-oriented programming language, comprising the steps of:

obtaining a set of classes, each of said classes being classified as one of mutable, immutable, and undecided;

testing each undecided class, said test being comprised of the sub-steps of:

testing mutability of each instance field in said undecided class being tested;

classifying an instance field as immutable if no possible state or encapsulation analysis modifications are found;

classifying an instance field as undecided if there is insufficient class mutability information; and

classifying an instance field as mutable otherwise;

re-classifying an undecided class as mutable if any instance fields in said undecided class are mutable;

re-classifying said undecided class as immutable if all instance fields in said undecided class are immutable;

repeating said testing each undecided class step until a number of undecided classes after a repetition of said testing step is identical to a number of undecided classes before the repetition of said testing step;

re-classifying remaining undecided classes as mutable classes; and

testing mutability of each class field in each class.

20. The method as recited in claim 19, wherein testing mutability of a field, whether said field is an instance field or a class field, is comprised of the sub-steps of:

determining whether any variable corresponding to said field being tested could undergo a state modification of a first type, said first type state modification being made by at least one method that is within said program component; and

performing encapsulation analysis to determine whether any variable corresponding to said field being tested could undergo a state modification of a second type, said second type state modification being made by at least one method that is not within said program component;

classifying said field being tested as immutable if no possible state modifications or breakages of encapsulation are found;

classifying said field being tested as undecided if there is insufficient class mutability information; and

classifying said field being tested as mutable otherwise.

21. The method as recited in claim 20, wherein the first type state modification determination sub-step comprises the steps of:

detecting possible first type state modification of a value held in said each variable; and

detecting possible first type state modification of a state of any object referenced by said each variable;

wherein a state of an object is modified if it can change after said object is initialized, and the state of an object is a set of states of all associated variables; and

wherein a variable is mutable if its state ever changes after said variable is initialized, the state of said variable being its value together with a state of any referenced object.

22. The method as recited in claim 20, wherein the performing encapsulation analysis sub-step comprises the steps of:

detecting possible second type state modification of a value of said each variable;

detecting possible second type state modification of a state of any object referenced by said each variable, said possible second type state modification of a state of any object occurring at a point of initialization; and

detecting possible breakage of variable encapsulation;

wherein a state of an object is modified if it can change after said object is initialized, and the state of an object is a set of states of all associated variables;

wherein a variable is mutable if its state ever changes after said variable is initialized, the state of said variable being its value together with a state of any referenced object;

wherein a variable is encapsulated if all references to objects reachable from it are defined within said component; and

wherein variable encapsulation is broken if a method within the program component causes a mutable object reachable from a variable to become accessible to at least one method that is not within said component.

23. The method as recited in claim 19, wherein the method is implemented in a Java environment, said instance fields being non-`static` fields, an instance variable being initialized upon completion of its corresponding `<init>`

method, and said class fields being `static` fields, a class variable being initialized upon completion of its corresponding `<clinit>` method.

24. The method as recited in claim 19, further comprising the steps of:

identifying an object as mutable if it is an instance of a mutable class;

identifying an object as immutable if it is an instance of an immutable class; and

identifying fields and objects that can be determined to be constants because said identified fields and objects are not in a set of detected mutable fields and objects.

25. The method as recited in claim 19, further comprising the step of:

identifying isolation faults due to detected mutable class fields.

26. A device for detecting mutability of variables, objects, fields, and classes in a program component, said component being written in an object-oriented programming language, comprising:

a layer of at least one core library and at least one data-flow analysis engine, for providing a particular abstraction of the program component;

a layer of at least one utility module, for using the results of the at least one data analysis engine to generate basic results; and

a layer of at least one mutability sub-analysis module, for generating final results;

wherein a variable is mutable if its state ever changes after said variable is initialized, the state of a variable being its value together with a state of any referenced object;

wherein an object is mutable if its state ever changes after said object is initialized, the state of an object being a set of states of all associated variables;

wherein a field is mutable if any variable corresponding to said field is mutable; and

wherein a class is mutable if any instance fields implemented by said class are mutable.

27. The device as recited in claim 26, the layer of at least one core library and at least one data analysis engine comprising:

a library for collecting and manipulating static information about the program component by analyzing a set of classfiles, and for effectively constructing the program component's reference, hierarchy, and call graphs.

28. The device as recited in claim 26, the layer of at least one core library and at least one data analysis engine comprising:

a library for allowing a user to read classfiles.

29. The device as recited in claim 26, the layer of at least one core library and at least one data analysis engine comprising:

an intra-procedural data analysis engine for iteratively computing an effect of an instruction on information associated with locations on a method frame, said method frame being an operand stack and a local variables array.

30. The device as recited in claim 26, the layer of at least one core library and at least one data analysis engine comprising:

an inter-procedural data analysis engine for computing the effect of a method on information associated with variables that still exist upon completion of this method.

31. The device as recited in claim 26, the layer of at least one utility module comprising:

a type analysis utility module for identifying a set of possible types for each instruction and each frame location in each method.

32. The device as recited in claim 26, the layer of at least one utility module comprising:

a reachability analysis utility module for identifying, for each method, a set of escaping objects and class variables from which a mutable object is reachable for each instruction and each frame location referring to said mutable object.

33. The device as recited in claim 26, the layer of at least one mutability sub-analysis module comprising:

a value modification mutability sub-analysis module for identifying, for each method, a set of fields whose corresponding instance and class variables may be set within said each method.

34. The device as recited in claim 26, the layer of at least one mutability sub-analysis module comprising:

an object modification mutability sub-analysis module for identifying, for each method, a set of reference-type fields and method parameters, said set of reference-type fields and method parameters referencing an object, a state of said object being modified by said each method.

35. The device as recited in claim 26, the layer of at least one mutability sub-analysis module comprising:

a variable accessibility mutability sub-analysis module for identifying, for each variable, whether its value may be modified directly by at least one method that is not within the program component.

36. The device as recited in claim 26, the layer of at least one mutability sub-analysis module comprising:

an object accessibility mutability sub-analysis module for detecting possible accessibility of a state of each object, by determining if each variable associated with said object is encapsulated;

wherein a variable is encapsulated if all references to objects reachable from it are defined within the program component; and

wherein said accessibility is made by at least one method that is not within the program component.

37. The device as recited in claim 26, the layer of at least one mutability sub-analysis module comprising:

a breakage of encapsulation mutability sub-analysis module for detecting a possible breakage of encapsulation;

wherein a variable is encapsulated if all references to mutable objects reachable from it are defined within the program component; and

wherein variable encapsulation is broken if a method within the program component causes a mutable object reachable from the variable to become accessible to at least one method that is not within the program component.

38. A computer system for detecting mutability of variables, objects, fields, and classes in a program component, said component being written in an object-oriented programming language, the computer system comprising:

at least one computer-readable memory including:

code that determines whether any variable in the program component could undergo a state modification of a first type, said first type state modification being made by at least one method that is within the program component; and

code that performs encapsulation analysis to determine whether any variable in the program component could undergo a state modification of a second type, said second type state modification being made by at least one method that is not within the program component;

wherein a variable is mutable if its state ever changes after said variable is initialized, the state of said variable being its value together with a state of any referenced object;

wherein an object is mutable if its state ever changes after said object is initialized, the state of said object being a set of states of all associated variables;

wherein a field is mutable if any variable corresponding to said field is mutable; and

wherein a class is mutable if any instance fields implemented by said class are mutable.

39. The computer system as recited in claim 38, wherein the code that determines whether any variable could undergo the first type state modification comprises:

code that detects possible first type state modification of a value held in said each variable; and

code that detects possible first type state modification of a state of any object referenced by said each variable.

40. The computer system as recited in claim 38, wherein the code that performs encapsulation analysis step comprises:

code that detects possible second type state modification of a value held in said each variable;

code that detects possible second type state modification of a state of any object referenced by said each variable, said possible second type state modification of a state of any object occurring at a point of initialization; and

code that detects possible breakage of variable encapsulation;

wherein a variable is encapsulated if all references to objects reachable from it are defined within the program component; and

wherein variable encapsulation is broken if a method within the program component causes a mutable object reachable from the variable to become accessible to at least one method that is not within the program component.

41. The computer system as recited in claim 38, wherein the program component is implemented in a Java environment, said any instance fields being non-`static` fields, said variables being class variables or instance variables, each of said class variables being initialized upon completion of its corresponding `<clinit>` method, and each of said instance variables being initialized upon completion of its corresponding `<init>` method.

42. The computer system as recited in claim 38, wherein the at least one computer-readable memory further includes:

code that identifyies isolation faults due to detected mutable global variables or objects.

43. The computer system as recited in claim 38, wherein the at least one computer-readable memory further includes:

code that identifies fields and objects that can be determined to be constants because said identified fields and objects are not in the set of detected mutable fields and objects.

44. A computer system for detecting mutability of variables, objects, fields, and classes in a program component, said component being written in an object-oriented programming language, the computer system comprising:

at least one computer-readable memory including:

code that obtains a set of classes, each of said classes being classified as one of mutable, immutable, and undecided;

code that tests each undecided class, said test being comprised of:

code that tests each field in said undecided class being tested, said field testing code being comprised of:

code that determines whether any variable corresponding to said each field could undergo a state modification of a first type, said first type state modification being made by at least one method that is within the program component; and

code that performs encapsulation analysis to determine whether any variable corresponding to said each field could undergo a state

modification of a second type, said second type state modification being made by at least one method that is not within the program component;

code that classifies said each field as immutable if no possible state modifications or breakages of encapsulation are found;

code that classifies said each field as mutable if possible state modifications or breakages of encapsulation are found; and

code that classifies said each field as undecided if there is insufficient class mutability information;

code that re-classifies said undecided class as mutable if any fields in said undecided class are mutable;

code that re-classifies said undecided class as immutable if all fields in said undecided class are immutable;

code that repeats said testing each undecided class code until a number of undecided classes after a repetition of said testing code is identical to a number of undecided classes before the repetition of said testing code; and

code that re-classifies remaining undecided classes as mutable classes.

45. A computer system for detecting mutability of variables, objects, fields, and classes in a program component, said component being written in an object-oriented programming language, the computer system comprising:

at least one computer-readable memory including:

code that obtains a set of classes, each of said classes being classified as one of mutable, immutable, and undecided;

code that tests each undecided class, said test being comprised of:

code that tests each instance field in said undecided class being tested, said instance field testing code being comprised of:

code that determines whether any variable corresponding to said each instance field could undergo a state modification of a first type, said first type state modification being made by at least one method that is within the program component; and

code that performs encapsulation analysis to determine whether any variable corresponding to said each instance field could undergo a state modification of a second type, said second type state modification being made by at least one method that is not within the program component;

code that classifies said each instance field as immutable if no possible state modifications or breakages of encapsulation are found;

code that classifies said each instance field as mutable if possible state modifications or breakages of encapsulation are found; and

code that classifies said each instance field as undecided if there is insufficient class mutability information;

code that re-classifies said undecided class as mutable if any instance fields in said undecided class are mutable;

code that re-classifies said undecided class as immutable if all instance fields in said undecided class are immutable;

code that repeats said testing each undecided class code until a number of undecided classes after a repetition of said testing code is identical to a number of undecided classes before the repetition of said testing code; and

code that re-classifies remaining undecided classes as mutable classes.

46. The computer system as recited in claim 45, wherein the code that determines whether any variable could undergo a first type state modification comprises:

code that detects possible first type state modification of a value held in said each variable; and

code that detects possible first type state modification of a state of any object referenced by said each variable;

wherein a state of an object is modified if it can change after said object is initialized, and the state of an object is a set of states of all associated variables; and

wherein a variable is mutable if its state ever changes after said variable is initialized, the state of said variable being its value together with a state of any referenced object.

47. The computer system as recited in claim 45, wherein the code that performs encapsulation analysis comprises:

code that detects possible second type modification of a value of said each variable;

code that detects possible second type modification of a state of any object referenced by said each variable, said possible second type state modification of a state of any object occurring at a point of initialization; and

code that detects possible breakage of variable encapsulation;

wherein a state of an object is modified if it can change after said object is initialized, and the state of an object is a set of states of all associated variables;

wherein a variable is mutable if its state ever changes after said variable is initialized, the state of said variable being its value together with a state of any referenced object;

wherein a variable is encapsulated if all references to objects reachable from it are defined within said component; and

wherein variable encapsulation is broken if a method within the program component causes a mutable object reachable from a variable to become accessible to at least one method that is not within said component.

48. The computer system as recited in claim 45, wherein the program component is implemented in a Java environment, said each variable corresponding to said each instance field being a non-`static` variable, and each non-`static` variable being initialized upon completion of its corresponding `<init>` method.

49. The computer system as recited in claim 45, wherein the at least one computer-readable memory further includes:

code that identifies an object as mutable if it is an instance of a mutable class;

code that identifies an object as immutable if it is an instance of an immutable class; and

code that identifies fields and objects that can be determined to be constants because said identified fields and objects are not in a set of detected mutable fields and objects.

50. The computer system as recited in claim 45, wherein the at least one computer-readable memory further includes:

code that tests mutability of each undecided class field in each class.

51. The computer system as recited in claim 50, wherein the at least one computer-readable memory further includes:

code that identifies isolation faults due to detected mutable class fields.

52. The computer system as recited in claim 50, wherein the code that tests mutability of each undecided class field in each class comprises:

code that determines whether any variable corresponding to said each undecided class field could undergo a first type state modification; and

code that performs encapsulation analysis to determine whether any variable corresponding to said each undecided class field could undergo a second type state modification.

53. A computer system for detecting mutability of classes and class variables in a program component, said component being written in an object-oriented programming language, comprising:

at least one computer-readable memory including:

code that obtains a set of classes, each of said classes being classified as one of mutable, immutable, and undecided;

code that tests each undecided class, said test being comprised of:

code that tests each instance field in said undecided class being tested, said instance field testing code being comprised of:

code that determines whether any variable corresponding to said each instance field could undergo a state modification of a first type, said first type state modification being made by at least one method that is within the program component; and

code that performs encapsulation analysis to determine whether any variable corresponding to said each instance field could undergo a state modification of a second type, said second type state modification being made by at least one method that is not within the program component;

code that classifies said each instance field as immutable if no possible state modifications or breakages of encapsulation are found;

code that classifies said each instance field as mutable if possible state modifications or breakages of encapsulation are found; and

code that classifies said each instance field as undecided if there is insufficient class mutability information;

code that re-classifies said undecided class as mutable if any instance fields in said undecided class are mutable;

code that re-classifies said undecided class as immutable if all instance fields in said undecided class are immutable;

code that repeats said testing each undecided class code until a number of undecided classes after a repetition of said testing code is identical to a number of undecided classes before the repetition of said testing code;

code that re-classifies remaining undecided classes as mutable classes; and

code that tests mutability of each class field in each class.

54. The computer system as recited in claim 53, wherein the code that tests mutability of a field, whether said field is an instance field or a class field, is comprised of:

code that determines whether any variable corresponding to said field being tested could undergo a state modification of a first type, said first type state

modification being made by at least one method that is within said program component; and

code that performs encapsulation analysis to determine whether any variable corresponding to said field being tested could undergo a state modification of a second type, said second type state modification being made by at least one method that is not within said program component;

code that classifies said field being tested as immutable if no possible state modifications or breakages of encapsulation are found;

code that classifies said field being tested as undecided if there is insufficient class mutability information; and

code that classifies said field being tested as mutable otherwise.

55. The computer system as recited in claim 54, wherein the code that determines whether any variable could undergo first type state modification comprises:

code that detects possible first type state modification of a value held in said each variable; and

code that detects possible first type state modification of a state of any object referenced by said each variable;

wherein a state of an object is modified if it can change after said object is initialized, and the state of an object is a set of states of all associated variables; and

wherein a variable is mutable if its state ever changes after said variable is initialized, the state of said variable being its value together with a state of any referenced object.

56. The computer system as recited in claim 54, wherein the code that performs encapsulation analysis comprises:

code that detects possible second type state modification of a value of said each variable;

code that detects possible second type state modification of a state of any object referenced by said each variable, said possible second type state modification of a state of any object occurring at a point of initialization; and

code that detects possible breakage of variable encapsulation;

wherein a state of an object is modified if it can change after said object is initialized, and the state of an object is a set of states of all associated variables;

wherein a variable is mutable if its state ever changes after said variable is initialized, the state of said variable being its value together with a state of any referenced object;

wherein a variable is encapsulated if all references to objects reachable from it are defined within said component; and

wherein variable encapsulation is broken if a method within the program component causes a mutable object reachable from a variable to become accessible to at least one method that is not within said component.

57. The computer system as recited in claim 53, wherein the program component is implemented in a Java environment, said instance fields being non-`static` fields, an instance variable being initialized upon completion of its corresponding `<init>` method, and said class fields being `static` fields, a class variable being initialized upon completion of its corresponding `<clinit>` method.

58. The computer system as recited in claim 53, wherein the at least one computer-readable memory further includes:

code that identifies an object as mutable if it is an instance of a mutable class;

code that identifies an object as immutable if it is an instance of an immutable class; and

code that identifies fields and objects that can be determined to be constants because said identified fields and objects are not in a set of detected mutable fields and objects.

59. The computer system as recited in claim 53, wherein the at least one computer-readable memory further includes:

code that identifies isolation faults due to detected mutable class fields.

60. A computer system for detecting mutability of variables, objects, fields, and classes in a program component, said component being written in an object-oriented programming language, the computer system comprising:

at least one computer-readable memory including:

code that maintains a layer of at least one core library and at least one data-flow analysis engine in a mutability analyzer, for providing a particular abstraction of the program component;

code that maintains a layer of at least one utility module in a mutability analyzer, for using the results of the at least one data analysis engine to generate basic results; and

code that maintains a layer of at least one mutability sub-analysis module in a mutability analyzer, for generating final results;

wherein a variable is mutable if its state ever changes after said variable is initialized, the state of said variable being its value together with a state of any referenced object;

wherein an object is mutable if its state ever changes after said object is initialized, the state of said object being a set of states of all associated variables;

wherein a field is mutable if any variable corresponding to said field is mutable; and

wherein a class is mutable if any instance fields implemented by said class are mutable.

61. The computer system as recited in claim 60, wherein the code that maintains the layer of at least one core library and at least one data analysis engine comprises:

code that collects and manipulates static information about the program component by analyzing a set of classfiles; and

code that effectively constructs the program component's reference, hierarchy, and call graphs.

62. The computer system as recited in claim 60, wherein the code that maintains the layer of at least one core library and at least one data analysis engine comprises:

code that allows a user to read classfiles.

63. The computer system as recited in claim 60, wherein the code that maintains the layer of at least one core library and at least one data analysis engine comprises:

code that iteratively computes an effect of an instruction on information associated with locations on a method frame, said method frame being an operand stack and a local variables array.

64. The computer system as recited in claim 60, wherein the code that maintains the layer of at least one core library and at least one data analysis engine comprises:

code that computes the effect of a method on information associated with variables that still exist upon completion of this method.

65. The computer system as recited in claim 60, wherein the code that maintains the layer of at least one utility module comprises:

code that identifies a set of possible types for each instruction and each frame location in each method.

66. The computer system as recited in claim 60, wherein the code that maintains the layer of at least one utility module comprises:

code that identifies, for each method, a set of escaping objects and class variables from which a mutable object is reachable for each instruction and each frame location referring to said mutable object.

67. The computer system as recited in claim 60, wherein the code that maintains the layer of at least one mutability sub-analysis module comprises:

code that identifies, for each method, a set of fields whose corresponding instance and class variables may be set within said each method.

68. The computer system as recited in claim 60, wherein the code that maintains the layer of at least one mutability sub-analysis module comprises:

code that identifies, for each method, a set of reference-type fields and method parameters, said set of reference-type fields and method parameters referencing an object, a state of said object being modified by said each method.

69. The computer system as recited in claim 60, wherein the code that maintains the layer of at least one mutability sub-analysis module comprises:

code that identifies, for each variable, whether its value may be modified directly by at least one method that is not within the program component.

70. The computer system as recited in claim 60, wherein the code that maintains the layer of at least one mutability sub-analysis module comprises:

code that detects possible accessibility of a state of each object, by determining if each variable associated with said object is encapsulated;

wherein a variable is encapsulated if all references to objects reachable from

it are defined within the program component; and

wherein said accessibility is made by at least one method that is not within the program component.

71. The computer system as recited in claim 60, wherein the code that maintains the layer of at least one mutability sub-analysis module comprises:

code that detects a possible breakage of encapsulation;

wherein a variable is encapsulated if all references to mutable objects reachable from it are defined within the program component; and

wherein variable encapsulation is broken if a method within the program component causes a mutable object reachable from the variable to become accessible to at least one method that is not within the program component.